Decoupled Quorum-based Byzantine-Resilient Coordination in Open Distributed Systems

Alysson Neves Bessani, Miguel Correia,
Joni da Silva Fraga, Lau Cheuk Lung

DI–FCUL TR–07–9

May 2007

Departamento de Informática
Faculdade de Ciências da Universidade de Lisboa
Campo Grande, 1749–016 Lisboa
Portugal

Technical reports are available at `http://www.di.fc.ul.pt/tech-reports`. The files are stored in PDF, with the report number as filename. Alternatively, reports are available by post from the above address.

Decoupled Quorum-based Byzantine-Resilient Coordination in Open Distributed Systems*

Alysson Neves Bessani† Miguel Correia† Joni da Silva Fraga‡ Lau Cheuk Lung‡

† LASIGE, Faculdade de Ciências da Universidade de Lisboa – Portugal

‡ Universidade Federal de Santa Catarina – Brazil

Abstract

Open distributed systems are typically composed by an unknown number of processes running in heterogeneous hosts. Their communication often requires tolerance to temporary disconnections and security against malicious actions. Tuple spaces are a well-known coordination model for this kind of systems. They can support communication that is decoupled both in time and space. There are currently several implementations of distributed fault-tolerant tuple spaces but they are not Byzantine-resilient, i.e., they do not provide a correct service if some replicas are attacked and start to misbehave. This paper presents an efficient implementation of LBTS, a Linearizable Byzantine fault-tolerant Tuple Space. LBTS uses a novel Byzantine quorum systems replication technique in which most operations are implemented by quorum protocols while stronger operations are implemented by more expensive protocols based on consensus. LBTS is linearizable and wait-free, showing interesting performance gains when compared to a similar construction based on state machine replication.

Keywords: Tuple Spaces, Byzantine Fault Tolerance, Intrusion Tolerance, Quorum Systems.

1 Introduction

Coordination is a classical distributed systems paradigm based on the idea that separating the system activities in computation and coordination can simplify the design of distributed applications [22]. The *generative coordination model*, originally introduced in the LINDA programming language [21], uses a shared memory object called a *tuple space* to support the coordination. Tuple spaces can support communication that is decoupled both in time – processes do not have to be active at the same time – and space – processes do not need to know each others' addresses [9]. The tuple space can be considered to be a kind of storage that stores *tuples*, i.e., finite sequences of values. The operations supported are essentially three: inserting a tuple in the space, reading a tuple from the space and removing a tuple from the space. The programming model supported

*This report was submitted for publication.

by tuple spaces is regarded as simple, expressive and elegant, being implemented in middleware platforms like GIGASPACES [24], Sun's JAVASPACES [40] and IBM's TSPACES [41].

There has been some research about fault-tolerant tuple spaces (e.g., [3, 43]). The objective of those works is essentially to guarantee the availability of the service provided by the tuple space, even if some of the servers that implement it crash. This paper goes one step further by describing a tuple space that tolerates Byzantine faults. More specifically, this work is part of a recent research effort in *intrusion-tolerant systems*, i.e., on systems that tolerate malicious faults, like attacks and intrusions [20, 42]. These faults can be modeled as arbitrary faults, also called Byzantine faults [27] in the literature.

The proposed tuple space is dubbed LBTS since it is a *Linearizable Byzantine Tuple Space*. LBTS is implemented by a set of distributed servers and behaves according to its specification if up to a number of these servers fail, either accidentally (e.g., crashing) or maliciously (e.g., by being attacked and starting to misbehave). Moreover, LBTS also tolerates accidental and malicious faults in an unbounded number of clients accessing it.

Although this is the first linearizable Byzantine tuple space that we are aware of, there are several domains in which it might be interesting to use this service. One case are application domains with frequent disconnections and mobility that can benefit from the time and space decoupling provided by LBTS. Two examples of such domains are *ad hoc networks* [35] and *mobile agents* [9]. Another domain are bag-of-tasks applications in *grid computing* [18], where a large number of computers are used to run complex computations. These applications are decoupled in space and time since the computers that run the application can enter and leave the grid dynamically.

LBTS has two important properties. First, it is *linearizable*, i.e., it provides a strong concurrency semantics in which operations invoked concurrently appear to take effect instantaneously sometime between their invocation and the return of their result [26]. Second, it is *wait-free*, i.e., every correct client process that invokes an operation in LBTS eventually receives a response, independently of the failure of other client processes or access contention [25].

Additionally, LBTS is based on a novel *Byzantine quorum systems* replication philosophy in which the semantics of each operation is carefully analyzed and a protocol as simple as possible is defined for each. Most operations on the tuple space are implemented by pure asynchronous

Byzantine quorum protocols [29]. However, a tuple space is a shared memory object with consensus number higher than one [39], according to Herlihy's wait-free hierarchy [25], so it cannot be implemented using only asynchronous quorum protocols [17]. In this paper we identify the tuple space operations that require stronger protocols, and show how to implement them using a *Byzantine* PAXOS consensus protocol [11, 32, 44]. The philosophy behind our design is that simple operations are implemented by "cheap" quorum-based protocols, while stronger operations are implemented by more expensive protocols based on consensus. These protocols are more expensive in two senses. First, they have higher communication and message complexities (e.g., the communication complexity is typically $O(n^2)$ instead of $O(n)$). Second, while Byzantine quorum protocols can be strictly asynchronous, consensus has been shown to be impossible to solve deterministically in asynchronous systems [19], so additional assumptions are needed: either about synchrony or about the existence of random oracles. Although there are other recent works that use quorum-based protocols to implement objects stronger than atomic registers [1] and to optimize state machine replication [14], LBTS is the first to mix these two approaches supporting wait freedom and being efficient even in the presence of contention.

Summary of Contributions. The main contributions of the paper are the following:

1. it presents the first linearizable tuple space that tolerates Byzantine faults; the tuple space requires $n \geq 4f+1$ servers, from which f can be faulty, and tolerates any number of faulty clients. Moreover, it presents a variant of this design that requires the minimal number of servers ($n \geq 3f+1$) at the cost of having weaker semantics;

2. it introduces a new design philosophy to implement shared memory objects with consensus number higher than one [25], by using quorum protocols for the weaker operations and consensus protocols for stronger operations. To implement this philosophy several new techniques are developed;

3. it presents the correctness conditions for a linearizable tuple space; although this type of object has been used for more than two decades, there is no other work that provides such a formalization;

4. it compares the proposed approach with Byzantine state machine replication [11, 38] and shows that LBTS presents several benefits: some operations are much cheaper and it supports the concurrent execution of operations, instead of executing them in total order.

Paper Organization. The paper is organized as follows. Section 2 presents the background information for the paper as well as the definition of the system model assumed by our protocols. The definition of the correctness conditions for a linearizable tuple space is formalized in Section 3. The LBTS protocols are presented in Section 4. Section 5 presents several optimizations and improvements for the basic LBTS protocols. An alternative version of LBTS is presented in Section 6. Sections 7 and 8 presents an evaluation of LBTS and summarize the related work, respectively. The conclusions are presented in Section 9, and an Appendix containing the complete proofs of the protocols is included in the end.

2 Preliminaries

2.1 Tuple Spaces

The *generative coordination* model, originally introduced in the LINDA programming language [21], uses a shared memory object called a *tuple space* to support the coordination between processes. This object essentially allows the storage and retrieval of generic data structures called *tuples*.

Each tuple is a sequence of fields. A tuple t in which all fields have a defined value is called an *entry*. A tuple with one or more undefined fields is called a *template* (usually denoted by a bar, e.g., $\bar{t}$). An entry t and a template $\bar{t}$ *match* — $m(t,\bar{t})$ — if they have the same number of fields and all defined field values of $\bar{t}$ are equal to the corresponding field values of t. Templates are used to allow content-addressable access to tuples in the tuple space (e.g., template $\langle 1,2,* \rangle$ matches any tuple with three fields in which 1 and 2 are the values of the first and second fields, respectively).

A tuple space provides three basic operations [21]: $out(t)$ that outputs/inserts the entry t in the tuple space; $inp(\bar{t})$ that reads and removes some tuple that matches $\bar{t}$ from the tuple space; $rdp(\bar{t})$ that reads a tuple that matches $\bar{t}$ without removing it from the space. The *inp* and *rdp* operations are *non-blocking*, i.e., if there is no tuple in the space that matches the template, an error code is

returned. Most tuple spaces also provide blocking versions of these operations, *in* and *rd*. These operations work in the same way of their non-blocking versions but stay blocked until there is some matching tuple available.

These few operations together with the content-addressable capabilities of generative coordination provide a simple and powerfull programming model for distributed applications [5, 21, 41]. The drawback of this model is that it depends on an infrastructure object (the tuple space), which is usually implemented as a centralized server, being a single point of failure, the main problem addressed in this paper.

2.2 System Model

The system is composed by an infinite set of *client processes*[1] $\Pi = \{p_1, p_2, p_3, ...\}$ which interact with a set of *n servers* $U = \{s_1, s_2, ..., s_n\}$ that implements a tuple space with certain dependability properties. We consider that each client process and each server has an unique id.

All communication between client processes and servers is made over *reliable authenticated point-to-point channels*[2]. All servers are equipped with a local clock used to compute message timeouts. These clocks are not synchronized so their values can drift.

In terms of failures, we assume that an arbitrary number of client processes and a bound of up to $f \leq \lfloor \frac{n-1}{4} \rfloor$ servers can be subject to *Byzantine failures*, i.e., they can deviate arbitrarily from the algorithm they are specified to execute and work in collusion to corrupt the system behavior. Clients or servers that do not follow their algorithm in some way are said to be *faulty*. A client/server that is not faulty is said to be *correct*. We assume *fault independence* for the servers, i.e., that the probability of each server failing is independent of another server being faulty. This assumption can be substantiated in practice using several kinds of diversity [12, 36].

We assume an *eventually synchronous system model* [16]: in all executions of the system, there is a bound Δ and an instant GST (Global Stabilization Time), so that every message sent by a correct server to another correct server at instant $u >$ GST is received before $u + \Delta$. Δ and GST are unknown. The intuition behind this model is that the system can work asynchronously (with no bounds on delays) most of the time but there are stable periods in which the communication delay

[1] We also call a client process simply *client* or *process*.

[2] These channels can easily be implemented in practice assuming fair links and using retransmissions, or in a more practical view, using TCP over IPsec or SSL/TLS.

is bounded (assuming local computations take negligible time)[3]. This assumption of eventually synchrony is needed to guarantee the termination of the Byzantine PAXOS [11, 32, 44]. An execution of a distributed algorithm is said to be *nice* if the bound Δ always holds and there are no server failures.

Additionally, we use a *digital signature scheme* that includes a signing function and a verification function that use pairs of public and private keys [37]. A message is signed using the signing function and a private key, and this signature is verified with the verification function and the corresponding public key. We assume that each correct server has a private key known only by itself, and that its public key is known by all client processes and servers. We represent a message signed by a server s with a subscript σ_s, e.g., m_{σ_s}.

2.3 Byzantine Quorum Systems

Quorum systems are a technique for implementing dependable shared memory objects in message passing distributed systems [23]. Given a universe of data servers, a quorum system is a set of server sets, called *quorums*, that have a non-empty intersection. The intuition is that if, for instance, a shared variable is stored replicated in all servers, any read or write operation has to be done only in a quorum of servers, not in all servers. The existence of intersections between the quorums allows the development of read and write protocols that maintain the integrity of the shared variable even if these operations are performed in different quorums.

Byzantine quorum systems are an extension of this technique for environments in which client processes and servers can fail in a Byzantine way [29]. Formally, a Byzantine quorum system is a set of server quorums $\mathcal{Q} \subseteq 2^U$ in which each pair of quorums intersect in sufficiently many servers (*consistency*) and there is always a quorum in which all servers are correct (*availability*).

The servers can be used to implement one or more shared memory objects. In this paper the servers implement a single object – a tuple space. The servers form a *f-masking quorum system*, which tolerates at most f faulty servers, i.e., it masks the failure of at most that number of servers [29]. This type of Byzantine quorum systems requires that the majority of the servers in the intersection between any two quorums are correct, thus $\forall Q_1, Q_2 \in \mathcal{Q}, |Q_1 \cap Q_2| \geq 2f+1$. Given this requirement, each quorum of the system must have $q = \lceil \frac{n+2f+1}{2} \rceil$ servers and the quorum

[3] In practice this stable period has to be long enough for the algorithm to terminate, but does not need to be forever.

system can be defined as: $\mathscr{Q} = \{Q \subseteq U : |Q| = q\}$. This implies that $|U| = n \geq 4f+1$ servers [29]. With these constraints, a quorum system with $n = 4f+1$ will have quorums of size $3f+1$.

2.4 Byzantine PAXOS

Since LBTS requires some modifications to the basic Byzantine PAXOS total order protocol [11], this section briefly presents this protocol.

The protocol begins with a client sending a signed message m to all servers (see Figure 1). One of the servers, called the leader, is responsible for ordering the messages sent by the clients. The leader then sends a PRE-PREPARE message to all servers giving a sequence number i to m. A server accepts a PRE-PREPARE message if the proposal of the leader is *good*: the signature of m verifies and no other PRE-PREPARE message was accepted for sequence number i. When a server accepts a PRE-PREPARE message, it sends a PREPARE message with m and i to all servers. When a server receives $\lceil \frac{n+f}{2} \rceil$ PREPARE messages with the same m and i, it marks m as prepared and sends a COMMIT message with m and i to all servers. When a server receives $\lceil \frac{n+f}{2} \rceil$ COMMIT messages with the same m and i, it commits m, i.e., accepts that message m is the i-th message to be delivered.

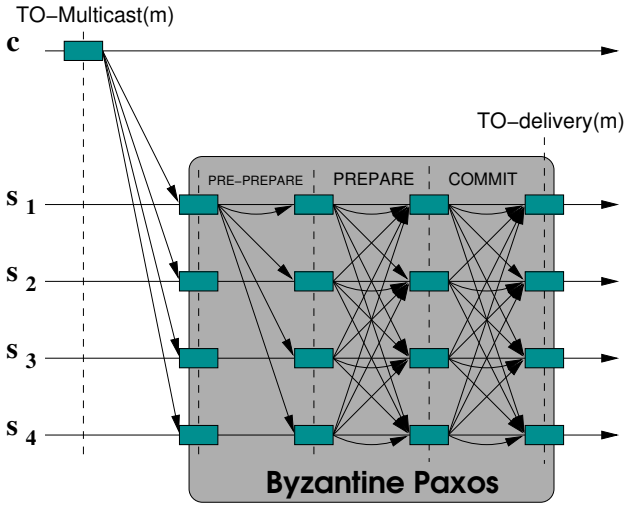

Figure 1: Byzantine PAXOS total order multicast execution.

While the PREPARE phase of the protocol ensures that there cannot be two prepared messages for the same sequence number i (which is sufficient to order messages when the leader is correct), the COMMIT phase ensures that a message committed with sequence number i will have this sequence number even if the leader is faulty.

When the leader is detected to be faulty, a leader election protocol is used to freeze the current round of the protocol, elect a new leader and start a new round. When a new leader is elected, it collects the protocol state from $\lceil \frac{n+f}{2} \rceil$ servers. The protocol state comprises information about accepted, prepared and committed messages. This information is signed and allows the new leader to verify if some message was already committed with some sequence number. Then, the new leader continues to order messages.

3 Tuple Space Correctness Conditions

Informally, a tuple space has to provide the semantics presented in Section 2.1. Here we specify this semantics formally using the notion of *history* [26].

A history H models an execution of a concurrent system composed by a set of processes and a shared memory object (a tuple space in our case). A history is a finite sequence of operation invocation events and operation response events. A *subhistory* S of a history H is a subsequence of the events of H.

We specify the properties of a tuple space in terms of *sequential histories*, i.e., histories in which the first event is an invocation and each invocation is directly followed by the corresponding response (or an event that signals the operation completion). We represent a sequential history H by a sequence of pairs $\langle operation, response \rangle$ separated by commas. We also separate subhistories by commas to form a new (sub)history. We use the membership operation $\in$ to mean "is a subhistory of".

A set of histories is said to be *prefix-closed* if H being in the set implies every prefix of H is also in the set. A *sequential specification* for an object is a prefix-closed set of sequential histories of that object.

A *sequential specification for a tuple space* is a set of prefix-closed histories of a tuple space in which any history H and any subhistory S of H satisfy the following properties[4]:

1. $S, \langle rdp(\bar{t}), t \rangle \in H \;\Rightarrow\; \exists \langle out(t), ack \rangle \in S$

2. $S, \langle rdp(\bar{t}), t \rangle \in H \;\Rightarrow\; \nexists \langle inp(\bar{t}'), t \rangle \in S$

[4]These properties also specify the blocking operations, substituting *inp*/*rdp* respectively by *in*/*rd*.

3. $S, \langle inp(\bar{t}), t\rangle \in H \;\Rightarrow\; \exists \langle out(t), ack\rangle \in S$

4. $\langle inp(\bar{t}), t\rangle, S \in H \;\Rightarrow\; \nexists \langle inp(\bar{t}'), t\rangle \in S$

The first property states that if a tuple t is read at a given instant, it must have been written before (the response for $out(t)$ is an acknowledgment, *ack*). The other properties have the same structure. For simplicity, the properties assume that each tuple is unique, i.e., that it is inserted only once in the space.

The properties 1-4 presented above are sufficient to prove tuple space linearizability, however, other property, that we call *not match* is needed to define when non-blocking read operations (*rdp* or *inp*) can return $\perp$. This property states the following: (*No match*) the special value $\perp$ can only be returned as a result of $rdp(\bar{t})$ (or $inp(\bar{t})$) if there is no tuple that matchs $\bar{t}$ inserted before the operation or all these tuples were removed before the operation.

We give a sequential specification of a tuple space but we want these properties to be satisfied by LBTS even when it is accessed concurrently by a set of processes. We guarantee this is indeed the case by proving that LBTS is *linearizable* [26] (see Appendix). This property states, informally, that operations invoked concurrently appear to take effect instantaneously sometime between their invocation and the return of a response (those not invoked concurrently are executed in the order they are invoked). In other words, any concurrent history of LBTS is equivalent to a sequential history. One should note that the sequential specification given above considers each tuple individually. This is sufficient to ensure the linearizability of LBTS given the inherent non-determinism on tuple access of the tuple space coordination model [21] (e.g., two sucessive *rdp* using the same template do not need to result on the same tuple reading).

The five properties above are *safety* properties of a tuple space. The *liveness* property we are interested in providing is *wait-free termination* [25]: every correct client process that invokes a non-blocking tuple space operation eventually receives a response, independent from other client failures or tuple space contention.

4 Linearizable Byzantine Tuple Space

This section presents LBTS. We concentrate our discussion only in tuple space non-blocking operations.

4.1 Design Rationale and New Techniques

The design philosophy of LBTS is to use quorum-based protocols for read (*rdp*) and write (*out*) operations, and an agreement primitive for the read-remove operation (*inp*). The implementation of this philosophy requires the development of some new techniques, described in this section.

To better understand these techniques let us recall how basic quorum-based protocols work. Traditionally, the objects implemented by quorums are read-write registers [4, 10, 28, 29, 30, 33]. The state of a register in each replica is represented by its current value and a timestamp (a kind of "version number"). The write protocol usually consists in *(i.)* reading the register current timestamp from a quorum, *(ii.)* incrementing it, and *(iii.)* writing the new value with the new timestamp in a quorum (deleting the old value). In the read protocol, the standard procedure is *(i.)* reading the pair timestamp-value from a quorum and *(ii.)* applying some read consolidation rule such as *"the current value of the register is the one associated with the greater timestamp that appears* $f+1$ *times"* to define what is the current value stored in the register. To ensure register linearizability (a.k.a. atomicity) two techniques are usually employed: *write-backs* – the read value is written again in the system to ensure that it will be the result of subsequent reads (e.g., [28, 30]) – or the *listener communication pattern* – the reader registers itself with the quorum system servers for receiving updates on the register values until it receives the same register state (timestamp-value) from a quorum, ensuring that this state will be observed in subsequent reads (e.g., [4, 10, 33]).

In trying to develop a tuple space object using these techniques two differences between this object and a register were observed: *(1.)* the state of the tuple space (the tuples it contains) can be arbitrarily large and *(2.)* the *inp* operation cannot be implemented by read and write protocols due to the requirement that the same tuple cannot be removed by two concurrent operations. Difference *(1.)* turns difficult using timestamps for defining what is the current state of the space (the state can be arbitrarily large) while difference *(2.)* requires that concurrent *inp* operations are executed in total order by all servers. The challenge is how to develop quorum protocols for implementing an object that does not use timestamps for versioning and, at the same time, requires a total order protocol in one operation. To solve these problems, we developed three algorithmic techniques.

The first technique introduced in LBTS serves to avoid timestamps in a collection object (one that its state is composed by a set of items added to it): we partition the state of the tuple space in infinitely many simpler objects, the tuples, that have three states: not inserted, inserted, and removed. This means that when a process invokes a read operation, the space chooses the response from the set of matching tuples that are in the inserted state. So, it does not need the version (timestamp) of the tuple space, because the read consolidation rule is applied to tuples and not to the space state.

The second technique is the application of the listener communication pattern in the *rdp* operation, to ensure that the usual quorum reasoning (e.g., a tuple can be read if it appears in $f+1$ servers) can be applied in the system even in parallel with executions of Byzantine PAXOS for *inp* operations. In the case of a tuple space, the *inp* operation is the single read-write operation: '*if there is some tuple that match* $\bar{t}$ *on the space, remove it*'. The listener pattern is used to "fit" the *rdp* between the occurrence of two *inp* operations. As will be seen in Section 4.2.2, the listener pattern is not used to ensure linearizability (as in previous works [4, 10, 33]), but for capturing replicas' state between removals. Linearizability is ensured using write-backs.

The third technique is the modification of the Byzantine PAXOS algorithm to allow the leader to propose the order *plus* a candidate result for an operation, allowing the system to reach an agreement even when there is no state agreement between the replicas. This is the case when the tuple space has to select a tuple to be removed that is not present in all servers. Notice that, without this modification, two agreements would have to be executed: one to decide what *inp* would be the first to remove a tuple, in case of concurrency (i.e., to order *inp* requests), and another to decide which tuple would be the result of the *inp*.

Another distinguished feature of LBTS is the number of replicas it requires. The minimal number of replicas required for asynchronous Byzantine-resilient quorum and consensus protocols are $3f+1$ [7, 33][5]. However, LBTS requires $n \geq 4f+1$ replicas. $3f+1$ replicas imply a quorum system with self-verifiable data, which requires a cryptographic expensive two-step preparing phase for write protocols [28], or the use of timestamps for the tuple space, which requires two additional steps in the *out* protocol that are vulnerable to timestamp exhaustion attacks.

[5]Without using special components like in [13], or weakening the protocol semantics, like in [34], when only $n \geq 2f+1$ suffices.

Solving this kind of attack requires asymmetric cryptography [28], threshold cryptography [10] or $4f+1$ replicas [4]. Moreover, the use of f more replicas allows the use of authenticators [11] in some operations that require digital signatures (see Section 5). Concluding, we trade f more replicas for simplicity and enhanced performance.

4.2 Protocols

Additional assumptions. We adopt several simplifications to improve the presentation of the protocols. First, we assume that all tuples are unique. In practice this might be implemented by appending to each tuple its writer id and a sequence number generated by the writer. Second, we assume that any message that was supposed to be signed by a server s and is not correctly signed is simply ignored. Third, all messages carry nonces in order to avoid replay attacks. Fourth, access control is implicitly enforced: the tuple space has some kind of access control mechanism (like an ACL) specifying what processes can insert tuples in it and each tuple has two sets of processes that can read and remove it. Fifth, the algorithms are described considering a single tuple space T, but their extension to support multiple tuple spaces is straightforward: a copy of each space is deployed in each server and all protocols are executed in the scope of one of the spaces (adding a field in each message indicating which tuple space is being accessed). Finally, we assume that the reactions of the servers to message receptions are atomic, i.e., that they are not preempted (e.g., lines 3-6 in Algorithm 1).

Protocol variables. Before we delve into the protocols, we have to introduce four variables stored in each server s: T_s, r_s, R_s and L_s. T_s is the local copy of the tuple space T in this server. The variable r_s gives the number of tuples previously removed from the tuple space replica in s. The set R_s contains the tuples already removed from the local copy of the tuple space (T_s). We call R_s the removal set and we use it to ensure that a tuple is not removed more than once from T_s. Variable r_s is updated only by the *inp* protocol. Later, in Section 4.2.3, we see that this operation is executed in all servers in the same order. Therefore, the value of r_s follows the same sequence in all correct servers. In the basic protocol (without the improvements of Section 5) $r_s = |R_s|$. Finally, the set L_s contains all clients registered to receive updates from this tuple space. This set is used in the *rdp* operation (Section 4.2.2). The protocols use a function $send(to, msg)$ to send a message msg to the recipient to, and a function $receive(from, msg)$ to receive a message, where

from is the sender and *msg* the message received.

4.2.1 Tuple Insertion - *out*

Algorithm 1 presents the *out* protocol. This protocol allows multiple writers without timestamps.

Algorithm 1 *out* operation (client p and server s).

	{SERVER}
{CLIENT}	**upon** $receive(p, \langle \text{OUT}, t \rangle)$
procedure $out(t)$	3: **if** $t \notin R_s$ **then**
1: $\forall s \in U$, $send(s, \langle \text{OUT}, t \rangle)$	4: $T_s \leftarrow T_s \cup \{t\}$
2: **wait until** $\exists Q \in \mathcal{Q} : \forall s \in Q$, $receive(s, \langle \text{ACK-OUT} \rangle)$	5: **end if**
	6: $send(p, \langle \text{ACK-OUT} \rangle)$

When a process p wants to insert a tuple t in the tuple space, it sends t to all servers (line 1) and waits for acknowledgments from a quorum of servers[6] (line 2). At the server side, if the tuple is not in the removal set R_s (indicating that it has already been removed) (line 3), it is inserted in the tuple space (line 4). An acknowledgment is returned (line 6).

With this simple algorithm a faulty client process can inserts a tuple in a subset of the servers. In that case, we say that it is an *incompletely inserted tuple*. The number of incomplete insertions made by a process can be bounded to one, as described in Section 5. As can be seen in next sections, *rdp* (resp. *inp*) operations are able to read (resp. remove) such a tuple if it is inserted in $f+1$ servers.

Notice that this protocol is always *fast* (terminates in two communication steps) [15]. Additionally, this protocol is *confirmable* [34], i.e., a process executing *out* knows when the operation ends. Therefore, it provides an ordered semantics for *out*, which makes the coordination language provided by LBTS Turing powerful [8], i.e., all computable functions can be implemented using it.

4.2.2 Tuple Reading - *rdp*

rdp is implemented by the protocol presented in Algorithm 2. The protocol is more tricky than the previous one for two reasons. First, it employs the listener communication pattern to capture the replicas state between removals. Second, if a matching tuple is found, the process may have

[6] In fact, it would be possible to implement this step sending the message to a quorum and then, periodically, to other servers, until there are responses from a quorum.

to write it back to the system to ensure that it will be read in subsequent reads, satisfying the linearizability property.

Algorithm 2 *rdp* operation (client p and server s).

{CLIENT}
procedure $rdp(\bar{t})$
1: $\forall s \in U$, $send(s, \langle \text{RDP}, \bar{t} \rangle)$
2: $\forall x \in \{1,2,...\}, \forall s \in U, Replies[x][s] \leftarrow \perp$
3: **repeat**
4: **wait until** $receive(s, \langle \text{REP-RDP}, s, T_s^{\bar{t}}, r_s \rangle_{\sigma_s})$
5: $Replies[r_s][s] \leftarrow \langle \text{REP-RDP}, s, T_s^{\bar{t}}, r_s \rangle_{\sigma_s}$
6: **until** $\exists r \in \{1,2,...\}, \{s \in U : Replies[r][s] \neq \perp\} \in \mathcal{Q}$
7: {From now on r indicates the r of the condition above}
8: $\forall s \in U$, $send(s, \langle \text{RDP-COMPLETE}, \bar{t} \rangle)$
9: **if** $\exists t, count\_tuple(t, r, Replies[r]) \geq q$ **then**
10: **return** t
11: **else if** $\exists t, count\_tuple(t, r, Replies[r]) \geq f + 1$ **then**
12: $\forall s \in U$, $send(s, \langle \text{WRITEBACK}, t, Replies[r] \rangle)$
13: **wait until** $\exists Q \in \mathcal{Q} : \forall s \in Q$, $receive(s, \langle \text{ACK-WB} \rangle)$
14: **return** t
15: **else**
16: **return** $\perp$
17: **end if**

{SERVER}
upon $receive(p, \langle \text{RDP}, \bar{t} \rangle)$
18: $L_s \leftarrow L_s \cup \{\langle p, \bar{t} \rangle\}$
19: $T_s^{\bar{t}} \leftarrow \{t \in T_s : m(t, \bar{t})\}$
20: $send(p, \langle \text{REP-RDP}, s, T_s^{\bar{t}}, r_s \rangle_{\sigma_s})$
upon $receive(p, \langle \text{RDP-COMPLETE}, \bar{t} \rangle)$
21: $L_s \leftarrow L_s \setminus \{\langle p, \bar{t} \rangle\}$
upon $receive(p, \langle \text{WRITEBACK}, t, proof \rangle)$
22: **if** $count\_tuple(t, proof) \geq f + 1$ **then**
23: **if** $t \notin R_s$ **then**
24: $T_s \leftarrow T_s \cup \{t\}$
25: **end if**
26: $send(p, \langle \text{ACK-WB} \rangle)$
27: **end if**
upon removal of t from T_s or insertion of t in T_s
28: **for all** $\langle p, \bar{t} \rangle \in L_s : m(t, \bar{t})$ **do**
29: $T_s^{\bar{t}} \leftarrow \{t' \in T_s : m(t', \bar{t})\}$
30: $send(p, \langle \text{REP-RDP}, s, T_s^{\bar{t}}, r_s \rangle_{\sigma_s})$
31: **end for**

Predicate: $count\_tuple(t, r, msgs) \triangleq |\{s \in U : msgs[s] = \langle \text{REP-RDP}, s, T_s^{\bar{t}}, r \rangle_{\sigma_s} \wedge t \in T_s^{\bar{t}}\}|$

When $rdp(\bar{t})$ is called, the client process p sends the template $\bar{t}$ to the servers (line 1). When a server s receives this message, it registers p as a listener, and replies with all tuples in T_s that match $\bar{t}$ [7] and the current number of tuples already removed r_s (lines 18-20). While p is registered as a listener, whenever a tuple is added or removed from the space a set with the tuples that match $\bar{t}$ is sent to p [8] (lines 28-31).

Process p collects replies from the servers, putting them in the *Replies* matrix, until it manages to have a set of replies from a quorum of servers reporting the state after the same number of tuple removals r (lines 2-6)[9]. After that, a RDP-COMPLETE message is sent to the servers (line 8).

The result of the operation depends on a single row r of the matrix *Replies*. This row represents a cut on the system state in which a quorum of servers processed exactly the same r removals, so, in this cut, quorum reasoning can be applied. This mechanism is fundamental to ensure that agreement algorithms and quorum-based protocols can be used together for different operations,

[7] If there are many tuples, only a given number of the oldest tuples are sent, and the client can request more as needed.

[8] In practice, only the update is sent to p.

[9] We use a matrix in the algorithm just to simplify the exposition. In practice this matrix is very sparse so it would have to be implemented using some other structure, like a list.

one of the novel ideas of this paper. If there is some tuple t in $Replies[r]$ that was replied by all servers in a quorum, then t is the result of the operation (lines 9-10). This is possible because this quorum ensures that the tuple can be read in all subsequent reads, thus ensuring linearizability. On the contrary, if there is no tuple replied by an entire quorum, but there is still some tuple t returned by more than f servers[10] for the same value of r, then t is *write-back* in the servers (line 11-12). The purpose of this write-back operation is to ensure that if t has not been removed until r, then it will be readable by all subsequent $rdp(\bar{t})$ operations requested by any client, with $m(t,\bar{t})$ and until t is removed. Therefore, the write-back is necessary to handle incompletely inserted tuples.

Upon the reception of a write-back message $\langle \text{WRITEBACK}, t, proof \rangle$, server s verifies if the write-back is *justified*, i.e., if *proof* includes at least $f+1$ correctly signed REP-RDP messages from different servers with r and t (line 22). A write-back that is not justified is ignored by correct servers. After this verification, if t is not already in T_s and has not been removed, then s inserts t in its local tuple space (lines 23-24). Finally, s sends a ACK-WB to the client (line 26), which waits for these replies from a quorum of servers and returns t (lines 13-14).

4.2.3 Tuple Destructive Reading - *inp*

The previous protocols are implemented using only Byzantine quorum techniques. The protocol for *inp*, on the other hand, requires stronger abstractions. This is a direct consequence of the tuple space semantics that does not allow *inp* to remove the same tuple twice (once removed it is no longer available). This is what makes the tuple space shared memory object have consensus number two [39, 25].

An approach to implement this semantics is to execute all *inp* operations in the same order in all correct servers. This can be made using a total order multicast protocol based on the Byzantine PAXOS algorithm (see Section 2.4). A simple approach would be to use it as an unmodified building block, but this requires two executions of the protocol for each *inp* [6]. To avoid this overhead, the solution we propose is based on *modifying* this algorithm in three specific points:

1. When the leader s receives a request $inp(\bar{t})$ from client p (i.e., a message $\langle \text{INP}, p, \bar{t} \rangle$), it

[10] If a tuple is returned by less than $f+1$ servers it can be a tuple that has not been inserted in the tuple space, created by a collusion of faulty servers.

sends to the other servers a PRE-PREPARE message with not only the sequence number i but also $\langle t_{\bar{t}}, \langle \text{INP}, p, \bar{t} \rangle_{\sigma_p} \rangle_{\sigma_s}$, where $t_{\bar{t}}$ is a tuple in T_s that matches $\bar{t}$. If there is no tuple that matches $\bar{t}$ in T_s, then $t_{\bar{t}} = \perp$.

2. A correct server s' accepts to remove the tuple $t_{\bar{t}}$ proposed by the leader in the PRE-PREPARE message if: *(i.)* the usual Byzantine PAXOS conditions for acceptance described in Section 2.4 are satisfied; *(ii.)* s' did not accept the removal of $t_{\bar{t}}$ previously; *(iii.)* $t_{\bar{t}}$ and $\bar{t}$ match; and *(iv.)* $t_{\bar{t}}$ is not forged, i.e., either $t \in T_s$ or s' received $f+1$ signed messages from different servers ensuring that they have t in their local tuple spaces. This last condition ensures that a tuple t can be removed if and only if it can be read, i.e., only if at least $f+1$ servers report having it.

3. When a new leader l' is elected, each server sends its protocol state to l' (as in the original total order Byzantine PAXOS algorithm[11]) and a signed set with the tuples in its local tuple space that match $\bar{t}$. This information is used by l' to build a proof for a proposal with a tuple t (in case it gets that tuple from $f+1$ servers). If there is no tuple reported by $f+1$ servers, this set of tuples justifies a $\perp$ proposal. This condition can be seen as a write-back from the leader in order to ensure that the tuple will be available in sufficiently many replicas before its removal.

Giving these modifications on the total order protocol, an *inp* operation is executed by Algorithm 3.

For a client p, the $inp(\bar{t})$ algorithm works exactly as if the replicated tuple space was implemented using state machine replication based on Byzantine PAXOS [11]: p sends a request to all servers and waits until $f+1$ servers reply with the same response, which is the result of the operation (lines 1-3).

In the server side, the requests for executions of *inp* received are inserted in the pending set P_s. When this set is not empty, the code in lines 4-13 is executed by the leader (the predicate $paxos\_leader(s)$ is *true* iff s is the current leader). For each pending request in P_s, a sequence number is attributed (line 5). Then, the leader picks a tuple from the tuple space that matches

[11]The objective is to ensure that a value decided by some correct server in some round (i.e., the request sequence number and the reply) will be the only possible decision in all subsequent rounds.

Algorithm 3 *inp* operation (client p and server s).

{CLIENT}
procedure $inp(\bar{t})$
1: $TO\text{-}multicast(U, \langle \text{INP}, p, \bar{t} \rangle)$
2: **wait until** receive $\langle \text{REP-INP}, t_{\bar{t}} \rangle$ from $f+1$ servers in U
3: **return** $t_{\bar{t}}$

{SERVER}
upon $paxos\_leader(s) \wedge P_s \neq \emptyset$
4: **for all** $\langle \text{INP}, p, \bar{t} \rangle \in P_s$ **do**
5: $i \leftarrow i+1$
6: **if** $\exists t \in T_s : m(t, \bar{t}) \wedge \neg marked(t)$ **then**
7: $t_{\bar{t}} \leftarrow t$
8: $mark(i, t)$
9: **else**
10: $t_{\bar{t}} \leftarrow \perp$
11: **end if**
12: $paxos\_propose(i, \langle t_{\bar{t}}, \langle \text{INP}, p, \bar{t} \rangle \rangle)$
13: **end for**

upon $paxos\_deliver(i, \langle t_{\bar{t}}, \langle \text{INP}, p, \bar{t} \rangle \rangle)$
14: $unmark(i)$
15: $P_s \leftarrow P_s \setminus \{\langle \text{INP}, p, \bar{t} \rangle\}$
16: **if** $t_{\bar{t}} \neq \perp$ **then**
17: **if** $t_{\bar{t}} \in T_s$ **then**
18: $T_s \leftarrow T_s \setminus \{t_{\bar{t}}\}$
19: **end if**
20: $R_s \leftarrow R_s \cup \{t_{\bar{t}}\}$
21: $r_s \leftarrow r_s + 1$
22: **end if**
23: $send(p, \langle \text{REP-INP}, t_{\bar{t}} \rangle)$

$\bar{t}$ (lines 6-7) and marks it with its sequence number to prevent it from being removed (line 8). The procedure $mark(i,t)$ marks the tuple as the one proposed to be removed in the i-th removal, while the predicate $marked(t)$ says if t is marked for removal. If no unmarked tuple matches $\bar{t}$, $\perp$ is proposed for the Byzantine PAXOS agreement (using the aforementioned PRE-PREPARE message), i.e., is sent to the other servers (lines 10, 12). The code in lines 4-13 corresponds to the modification 1 above. Modifications 2 and 3 do not appear in the code since they are reasonably simple changes on Byzantine PAXOS algorithm.

When the servers reach agreement about the sequence number and the tuple to remove, the *paxos_deliver* predicate is set to *true* and the lines 14-23 of the algorithm are executed. Then, each server s unmarks any tuple that it marked for removal with the sequence number i (line 14) and removes the ordered request from P_s (line 15). After that, if the result of the operation is a valid tuple $t_{\bar{t}}$, the server verifies if it exists in the local tuple space T_s (line 17). If it does, it is removed from T_s (line 18). Finally, $t_{\bar{t}}$ is added to R_s, the removal counter r_s is incremented and the result is sent to the requesting client process (lines 20-23).

It is worth noticing that Byzantine PAXOS usually does not employ public-key cryptography when the leader does not change. The signatures required by the protocol are made using *authenticators*, which are vectors of message authentication codes [11]. However, modification 3 requires that the signed set of tuples will be sent to a new leader when it is elected. Therefore, our *inp* protocol requires public-key cryptography, but only when the operation cannot be resolved in the first Byzantine PAXOS round execution.

4.3 Correctness

In this section we informaly argue about the correctness of the protocols. The complete proof of the arguments sketched here are presented in Appendix.

LBTS is linearizable because the tuple space correctness conditions (Section 3) imply that we only have to deal with concurrent operations that manipulate the same tuple. The algorithms were build in such a way that, if some process reads a tuple, the write-back phase of the *rdp* protocol ensures that this tuple will be available to all other *rdp* or *inp* operations until the tuple is removed.

The wait-freedom property is satisfied for all operations simply because the Byzantine PAXOS protocol always terminate in our (partially synchronous) system model. Therefore, both *rdp* (in which a client can have to wait until some removal terminate before collecting responses from a quorum of servers that executed the same removals) and *inp* (which comprises, basically, an execution of this protocol), will terminate.

5 Improvements and Optimizations

Optimizing *rdp*. The protocol in Algorithm 2 usually does not require the write-back phase when there are no faulty servers and there are many tuples in the space that match the requested template. In that case it is very likely that some tuple will be replied by a complete quorum of servers, thus avoiding the need for verifying if write-backs are justified, something that would imply verifying the signatures of a set of REP-RDP messages. Public-key cryptography has been shown to be a major bottleneck in practical Byzantine fault-tolerant systems [11], specially in LANs and other high speed networks, where the communication latency is lower and public-key cryptography processing costs dominate the operation latency. To avoid using public-key signatures we propose the following optimization on the *rdp* protocol: the client first accesses a quorum of servers asking for the tuples that match the template (without requiring signed REP-RDP messages from servers or the use of the listener pattern). If there is some tuple t returned by the whole quorum of servers, then the operation is finished and t is the result. If no tuple is returned by more than f servers then the operation is finished and the result is $\perp$. If some tuple is returned by at least $f+1$ servers, it means that possibly a write-back will be needed, so the protocol from Algorithm 2 is used. Notice that this optimization does not affect the correctness of the protocol

since the single delicate operation, the write-back, is executed using the normal protocol.

Bounding Memory. As described in Section 4.2, the R_s set stores all tuples removed from the space at a particular tuple space server s. In long executions, where a large number of tuples is inserted and removed from the tuple space, the R_s set will use an arbitrary amount of memory. Even considering that, in practice, we can bound the memory size by discarding old tuples from this set, since insertions and removals cannot take an infinite amount of time to complete[12], in theoretical terms, this requires unbounded memory in servers. Here, we present a scheme to discard tuples from the R_s sets.

This scheme is based on two modifications on the protocols. The *first need for the unbounded memory* is to avoid that clients executing *rdp* write-back a tuple which is being removed concurrently in the servers that already removed it (line 23 of Algorithm 2). In this way, the requirement is to prevent that tuples being removed are write-back. This can be implemented making each server s store the removed tuple t as long as there are *rdp* operations that started before the removal of t on s. More specifically, a removed tuple t must be stored in R_s until there are no clients in L_s (*listeners* set) that began their *rdp* operations in s (lines 1-17 of Algorithm 2) before the removal of t from s (lines 14-23 of Algorithm 3). In this way, the R_s set can be "cleaned" when s receives a message RDP-COMPLETE (line 21 of Algorithm 2).

The *second need for unbounded memory* on LBTS is avoiding more than one removal of the same tuple. This problem can happen if some tuple t is removed while its insertion is not complete, i.e., t is not in the local tuple space of a quorum of servers and is removed from the tuple space (recall that a tuple can be removed if it appears in at least $f+1$ servers). To solve this problem, we have to implement a control that ensures that a tuple that was not present in the *local tuple space* of server s (T_s) when it was removed from the *replicated tuple space* cannot be inserted in T_s after its removal (otherwise it could be removed more than once). More specifically, a removed tuple t will be removed from R_s only if t was received (inserted) by s.

To summarize, a tuple *can only be removed* from the R_s set when *(1.)* there is no *rdp* concurrent with the removal and *(2.)* when the tuple was already received by s.

These two modifications make the amount of tuples stored in the R_s set at a given time directly

[12]Recall that the removed tuples are stored in this set only to prevent them from being removed more than once during their insertion and/or removal.

dependent of the amount of *out* operations being executed concurrently with removals in s at this time (in the worst case, the concurrency of the system) and the number of partially inserted tuples in LBTS (that can be bounded at one per faulty client, as shown bellow).

Bounding Byzantine Clients. The LBTS protocols as described allow a malicious client to partially insert an unbounded number of tuples in a space. Here we present some modifications to the *out* protocol to bound to *one* the number of incomplete writes that a faulty client can make. The modifications are based on the use of *insertion certificates*: sets of signed ACK-OUT messages from a quorum of servers corresponding to the replies from an OUT message [28]. Each insertion has a timestamp provided by the client. Each server stores the greatest timestamp used by a client. To insert a new tuple a client must send an OUT message with timestamp greater than the last used by itself together with the insertion certificate of the previous completed write. A server accepts an insertion if the presented insertion certificate is valid, i.e., it contains at least $2f+1$ correctly signed ACK-OUT messages.

There are three main implications of this modification. First, it requires that all ACK-OUT messages are signed by the servers. However, we can avoid public key signature since we are assuming $n \geq 4f+1$ servers, and with this number of servers we can use authenticators [11], a.k.a. MAC vectors that implement signatures using only symmetric cryptography. In this way, an insertion certificate is valid for a server s if and only if it contains $q-f$ messages correctly authenticated for this server (the MAC field corresponding to s contains a correct signature). The second implication is the use of FIFO channels between clients and servers, which were not needed in the standard protocol. The final implication, also related with FIFO channels, is that each server must store the timestamp for the last write of a client to evaluate if the insertion certificate is correct, consequently, the amount of memory needed by the algorithm is proportional to the number of clients of the system[13].

Notice that this modification can be implemented in a layer bellow the LBTS *out* algorithm, without requiring any modification in it. Notice also that the use of authenticators adds very modest latency to the protocol [11].

[13] In practice this is not a problem, since only a single integer is require to be stored for each client.

6 Minimal LBTS

As stated in Section 4.1, LBTS requires a sub-optimal number of replicas ($n \geq 4f+1$) to implement a tuple space with confirmable semantics. However, it is possible to implement a tuple space with optimal resilience if we sacrifice the confirmation phase of some LBTS protocols. To implement LBTS with only $n \geq 3f+1$ we need to use an *asymmetric Byzantine quorum system* [34]. This type of quorum system has two types of quorums with different sizes: *read quorums* with $q_r = \lceil \frac{n+f+1}{2} \rceil$ servers and *write quorums* with $q_w = \lceil \frac{n+2f+1}{2} \rceil$.

The key property of this type of system is that every read quorum intersects every write quorum in at least $2f+1$ servers. This property makes this system very similar to the f-dissemination quorum system used by LBTS. Thus, the adaptation of LBTS to this kind of quorum is very simple: we need to make all write quorum operations (*out* protocol and write-back phase of the *rdp* protocol) non-confirmable, i.e., the request for write is sent to a write quorum and there is no wait for confirmation (ACK-OUT or ACK-WB messages).

In a non-confirmable protocol, the writer does not know when its write completes. In LBTS, an $out(t)$ operation ends when t is inserted in the local tuple space of $q_w - f$ correct servers. The same happens with the write-back phase of the *rdp* protocol. Missing this confirmation for an *out* operation has subtle but significant impacts on the tuple space computational power: if the protocol that implements *out* is non-confirmable, the resulting operation has *unordered* semantics and the coordination language provided by the tuple space will not be Turing powerfull [8].

The correctness proof of Minimal LBTS is the same of "normal" LBTS because the intersection between write and read quorums is at least $2f+1$ servers, which is the same intersection for the symmetrical quoruns used in LBTS.

7 Evaluation

This section presents an evaluation of the system using two distributed algorithms metrics: *message complexity* and *number of communication steps*. Message complexity measures the maximum amount of messages exchanged between processes, so it gives some insight about the communication system usage and the algorithm scalability. The number of communication steps is the number of sequential communications between processes, so usually it is the main factor for

the time needed for a distributed algorithm execution to terminate.

In this evaluation, we compare LBTS with an implementation of a tuple space with the same semantics based on *state machine replication* [38], which we call SMR-TS. SMR is a generic solution for the implementation of fault-tolerant distributed services using replication. The idea is to make all replicas start in the same state and deterministically execute the same operations in the same order in all replicas. The implementation considered for SMR-TS is based on the total order algorithm of [11] with the optimization for fast decision (two communication steps) in nice executions of [44, 32]. This optimization is also considered for the modified Byzantine PAXOS used in our *inp* protocol. The SMR-TS implements an optimistic version for read operations in which all servers return immediately the value read without executing the Byzantine PAXOS; the operation is successful in this optimistic phase if the process manages to collect $n-f$ identical replies (and perceives no concurrency), otherwise, the Byzantine PAXOS protocol is executed and the read result is the response returned by at least $f+1$ replicas. This condition ensures linearizability for all executions.

Operation	LBTS		SMR-TS	
	Message Complexity	Comm. Steps	Message Complexity	Comm. Steps
out	$\mathbf{O(n)}$	**2**	$\mathbf{O(n^2)}$	**4**
rdp	$\mathbf{O(n)}$	$\mathbf{2}/4$	$\mathbf{O(n)}/O(n^2)$	$\mathbf{2}/6$
inp	$\mathbf{O(n^2)}$	$\mathbf{4}/7$	$\mathbf{O(n^2)}$	**4**

Table 1: Costs in nice executions

Table 1 evaluates nice executions of the operations in terms of message complexity and communication steps[14]. The costs of LBTS' operations are presented in the second and third columns of the table. The fourth and fifth columns show the evaluation of SMR-TS. The LBTS protocol for *out* is cheaper than SMR-TS in both metrics. The protocol for *rdp* has the same costs in LBTS and SMR-TS in executions in which there is no matching tuple being written concurrently with *rdp*. The first values in the line of the table corresponding to *rdp* are about this optimistic case ($O(n)$ for message complexity, 2 for communication steps). When a read cannot be made optimistically, the operation requires 4 steps in LBTS and 6 in SMR-TS (optimistic phase plus the normal operation). Moreover LBTS' message complexity is linear, instead of $O(n^2)$ like SMR-TS. The protocol for *inp* uses a single Byzantine PAXOS execution in both approaches. However,

[14]Recall from Section 2.2 that an execution is said to be *nice* if the maximum delay Δ always hold and there are no failures.

in cases in which there are many tuples incompletely inserted (extreme contention or too many faulty clients), LBTS might not decide in the first round (as discussed in Section 5). In this case a new leader must be elected. We expect this situation to be rare. Notice that LBTS' quorum-based protocols (*rdp* and *inp*) are *fast* (terminates in two communication steps) when executed in favorable settings, matching the lower bound of [15].

The table allow us to conclude that an important advantage of LBTS when compared with SMR-TS is the fact that in SMR-TS all operations require protocols with message complexity $O(n^2)$, turning simple operations such as *rdp* and *out* as complex as *inp*. Another advantage of LBTS is that its quorum-based operations, *out* and *rdp*, always terminate in few communication steps while in SMR-TS these operation relies on Byzantine PAXOS, that we can have certainty that terminates in 4 steps only in nice executions [44, 32].

The evaluation of what happens in "not nice" situations is not shown in the table. In that case, all operations based on Byzantine PAXOS are delayed until there is enough synchrony for the protocol to terminate ($u > GST$). This problem is especially relevant in systems deployed in large scale networks and Byzantine environments in which an attacker might delay the communication at specific moments of the execution of the Byzantine PAXOS algorithm with the purpose of delaying its termination.

8 Related Work

Two replication approaches can be used to build Byzantine fault-tolerant services: Byzantine quorum systems [29] and state machine replication [38, 11]. The former is a data centric approach based on the idea of executing different operations in different intersecting sets of servers, while the latter is based on maintaining a consistent replicated state across all servers in the system. One advantage of quorum systems in comparison to the state machine approach is that they do not need that the operations are executed in the same order in the replicas, so they do not need to solve consensus. Quorum protocols usually scale much better due to the opportunity of concurrency in the execution of operations and the shifting of hard work from servers to client processes [1]. On the other hand, pure quorum protocols cannot be used to implement objects stronger than register (in asynchronous systems), on the contrary of state machine replication, which is more general

[17] (but, by the other hand, require aditional assumptions on the system).

The system presented in this paper uses a Byzantine quorum system and provides specific protocols for tuple space operations. Only one of the protocols (*inp*) requires a consensus protocol, therefore it needs more message exchanges and time assumptions (eventual synchrony) to terminate. This requirement is justified by the fact that tuple spaces have consensus number two [39] according to Herlihy's wait-free hierarchy [25], therefore they cannot be implemented deterministically in a wait-free way in a completely asynchronous system. To the best of our knowledge there is only one work on Byzantine quorums that has implemented objects more powerful than registers in a way that is similar to ours, the Q/U protocols [1]. That work aims to implement general services using quorum-based protocols in asynchronous Byzantine systems. Since this cannot be done ensuring wait-freedom, the approach sacrifices liveness: the operations are guaranteed to terminate only if there is no other operation executing concurrently. A tuple space build using Q/U has mainly two drawbacks, when compared with LBTS: *(i.)* it is not wait-free so, in a Byzantine environment, malicious clients could invoke operations continuously, causing a denial of service; and *(ii.)* it requires $5f+1$ servers, f more than LBTS, and it has an impact on the cost of the system due to the cost of diversity [36].

There are a few other works on Byzantine quorums related to ours. In [4], a non-skipping timestamps object is proposed. This type of object is equivalent to a *fetch&add* register, which is known to have consensus number two [25]. However, in order to implement this object in asynchronous systems using quorums, the specification is weakened in such a way that the resulting object has consensus number 1 (like a register). Some works propose consensus objects based on registers implemented using quorum protocols and randomization (e.g., [31]) or failure detectors (e.g., [2]). These works differ fundamentally from ours since they use basic quorum-based objects (registers) to build consensus while we use consensus to implement a more elaborated object (tuple space). Furthermore, the coordination algorithms provided in these works requires that processes know each other, a problem for open systems.

Cowling et al. proposed HQ-REPLICATION [14], an interesting replication scheme that uses quorum protocols when there are no contention in operations executions and consensus protocols to resolve contention situations. This protocol requires $n \geq 3f+1$ replicas and process reads and writes in 2 to 4 communication steps in contention-free executions. When contention is

detected, the protocol uses Byzantine PAXOS to order of contending requests. This contention resolution protocol adds great latency to the protocols, reaching more than 10 communication steps even in nice executions. Comparing LBTS with a tuple space based on HQ-REPLICATION, in executions without contention, LBTS' *out* will be faster (2 steps instead of 4 of HQ), *rdp* will be equivalent (the protocols are similar) and *inp* will have the same latency in both, however, LBTS' protocol has $O(n^2)$ message complexity instead of $O(n)$ of HQ. In contending executions, LBTS is expected to outperform HQ in orders of magnitude since its protocols are little affected by these situations. On the other hand, HQ-REPLICATION requires f fewer replicas than LBTS.

There are several works that replicate tuple spaces for fault tolerance. Some of them are based on the state machine approach (e.g., [3]) while others use quorum systems (e.g., [43]). However, none of these proposals deals with Byzantine failures and intrusions, the main objective of LBTS.

The construction presented in this paper, LBTS, builds on a preliminary solution with several limitations, BTS [6]. LBTS goes much further in mainly three aspects: it is linearizable; it uses a confirmable protocol for operation *out* (improving its semantical power [8]); and it implements the *inp* operation using only one Byzantine PAXOS execution, instead of two in BTS.

9 Conclusions

In this paper we presented the design of LBTS, a Linearizable Byzantine Tuple Space. This construction provides reliability, availability and integrity for coordination between processes in open distributed systems. The overall system model is based on a set of servers from which less than a fourth may be faulty and on an unlimited number of client processes, from which arbitrarily many can also be faulty. Given the time and space decoupling offered by the tuple space coordination model [9], this model appears to be an interesting alternative for coordination of non-trusted process in practical dynamic distributed systems (like P2P networks on the Internet or infrastructured wireless networks).

LBTS uses a novel hybrid replication technique which combines Byzantine quorum systems protocols with consensus-based protocols resulting in a design in which simple operations use simple quorum-based protocols while a more complicated operation, which requires servers's synchronization, uses more complex agreement-based protocols. The integration of these two

replication approaches required the development of some novel algorithmic techniques that are interesting by themselves. Concerning tuple space implementation, an important contribution of this work is the assertion that *out* and *rdp* can be implemented using quorum-based protocols, while *inp* requires consensus. This design shows important benefits when compared with the same object implemented using state machine replication.

Acknowledgements

We warmly thank Paulo Sousa, Piotr Zielinski and Rodrigo Rodrigues for their suggestions to improve the paper. This work was supported by LaSIGE, the EU through project IST-4-027513-STP (CRUTIAL) and CAPES/GRICES (project TISD).

Appendix: LBTS Correctness Proof

In this Appendix we prove that our protocols implement a tuple space that satisfies the correctness conditions stated in Section 3. We consider the protocols as presented in Section 4. In this section, when we say that a tuple *t exists* in some server *s*, or that server *s has t*, we mean that $t \in T_s$. Recall that there are $|U| = n \geq 4f+1$ servers and that the quorum size is $q = \lceil \frac{n+2f+1}{2} \rceil$ [15].

We begin by defining the notion of readable tuple.

Definition 1 *A tuple t is said to be* readable *if it would be the result of a* $rdp(\bar{t})$ *operation, with* $m(t,\bar{t})$*, executed without concurrency in a tuple space containing only the tuple t.*

The definition states that a tuple is readable if it would be the result of some *rdp* operation, independently of the accessed quorum or the number of failures in the system (but assuming less than $n/4$ servers fail). From the algorithm that implements *rdp*, line 11, it is simple to infer that a tuple is readable if it exists in $f+1$ correct servers in any quorum of the system. This implies that the tuple must exist in $n-q+f+1$ correct servers ($f+1$ servers from the quorum plus $n-q$ servers not from the quorum). The concept of readable tuple is used in the proofs to assert that if a tuple is correctly inserted, then it can be read (it is readable) or that if a tuple is correctly

[15] The proofs in this section are generic but we suggest the reader to use $n = 4f+1$ and $q = 3f+1$ to make them simpler to understand.

removed, then it cannot be read (it is not readable).

Given this definition, we state a lemma showing that if a tuple is readable then it can be removed.

Lemma 1 *If after an operation op a tuple t is readable, then t will be the result of* $inp(\bar{t})$ *executed immediately after op.*

Proof: Assume that after op the number of removed tuples is r. Assume a correct process p invokes $inp(\bar{t})$. If a tuple t is readable after r removals then there are at least $n-q+f+1$ correct servers that have t. In this condition, we have to prove that an execution of $inp(\bar{t})$ must return t. Considering that the current leader in the execution of the Byzantine PAXOS is the server l, we have to consider two cases:

1. *l is correct* – two cases have to be considered:

   (a) $t \in T_l$: in this case l will propose t to be the result of the operation (and no other tuple due to the definition of readable tuple). Since at least $n-q+f+1$ correct servers have t, they will accept this proposal and the tuple will be the result of the operation.

   (b) $t \notin T_l$: in this case l will propose $\perp$. This value will not be accepted because t exists in $n-q+f+1$ correct servers, so at most $q-f-1 < \lceil \frac{n+f}{2} \rceil$ servers can accept $\perp$ [16]. Therefore, a new leader is elected. Since no tuple was accepted for removal by $n-q+f+1$ servers, this leader will be able to choose t as the result. Any other proposal will be rejected and will cause a new leader election. Eventually a leader with $t \in T_l$ will be elected and case 1.(a) will apply.

2. *l is faulty* – in this case, l can propose $\perp$ or some $t' \neq t$. If $\perp$ is proposed, it will not be accepted by any of the $n-q+f+1$ correct servers that have t (because $m(t,\bar{t})$). If t' is proposed, it will not be accepted by more than f servers so it is not decided as the result of the operation. The reason for this is that, by the definition of readable, there is no other tuple in the space that matches $\bar{t}$, so no correct server will have t' in its local tuple space (or t' and $\bar{t}$ do not match). In both cases there will be at most $n-q+f$ servers that accept the result proposed by the leader and the decision will not be reached in this round, consequently a

[16] Recall that in the Byzantine PAXOS a value can be decided only if $\lceil \frac{n+f}{2} \rceil$ servers accept it (Section 2.4).

new leader will be elected. Depending on this leader being correct or not, cases 1 or 2 apply again.

In both cases, the result of the $inp(\bar{t})$ will be t. ■

The following lemma proves that a tuple cannot be read before being inserted in the space. This lemma is important because it shows that faulty servers cannot "create" tuples.

Lemma 2 *Before an $\langle out(t)/ack\rangle$, t is not readable.*

Proof: A tuple will be considered for reading if at least $f+1$ servers send it in reply to a read request. This means that at least one correct server must reply it. Since before $out(t)$ no correct server will reply the tuple t, then t is not readable. ■

Lemma 3 *After a $\langle out(t)/ack\rangle$ and before a $\langle inp(\bar{t})/t\rangle$, t is readable.*

Proof: The definition of readable considers that t is the only tuple in the tuple space that matches $\bar{t}$. If t has been inserted by an $out(t)$ then there is a quorum of servers Q_1 that have t. Consider a read operation performed after that insertion but before the removal of t from the space. We have to prove that t must be the result of a $rdp(\bar{t})$, assuming that $m(t,\bar{t})$. After line 6 of the $rdp(\bar{t})$ protocol, we know that a quorum Q_2 of servers replied the matching tuples they have after r removals. Every correct server of Q_2 that is member of Q_1 will reply t. Since the intersection of two quorums has at least $2f+1$ servers, $|Q_1 \cap Q_2| \geq 2f+1$, from which at most f are faulty, t will be returned by at least $f+1$ servers and will be the result of the operation. Therefore, t is readable. ■

The following lemma proves that when a tuple is read it remains readable until it is removed.

Lemma 4 *After a $\langle rdp(\bar{t})/t\rangle$ and before an $\langle inp(\bar{t})/t\rangle$, t is readable.*

Proof: We know that a quorum of servers has t after a $rdp(\bar{t})$ operation that returns t (due to the write-back phase). Therefore, following the same reasoning as in the proof of Lemma 3, we can infer that t is readable. ■

The following two lemmas prove that it is not possible to remove a tuple twice.

Lemma 5 *After a $\langle inp(\bar{t})/t\rangle$, t is not readable.*

Proof: Assume t was the r-th tuple removed from the space. Assume also, for sake of simplicity, that there are no more removals in the system. We will prove this lemma by contradiction. Suppose that a $rdp(\bar{t})$ operation executed after r removals returns t. This implies that t was reported by at least $f+1$ servers that have removed r tuples. This is clearly impossible because no correct server will report t after removing it. Consequently, the lemma is proved. ■

Lemma 6 *A tuple t cannot be removed more than once.*

Proof: Due to the Byzantine PAXOS total order algorithm safety properties, all removals are executed sequentially, one after another. The modification number 2 of this algorithm in Section 4.2.3 prevents correct servers from accepting for removal tuples already removed. ■

The following lemmas state that the three operations provided by LBTS satisfy wait-freedom [25], i.e., that they always terminate in our system model when invoked by a correct client.

Lemma 7 *Operation out is wait-free.*

Proof: An inspection of Algorithm 1 shows that the only place in which it can block is when waiting for replies from a quorum of servers (q servers) in line 2. Since all correct servers reply and $q \leq n-f$ (availability property in Section 2.3), the algorithm does not block. ■

Lemma 8 *Operation rdp is wait-free.*

Proof: In the first phase of Algorithm 2, client p waits for replies from a quorum of servers that removed r tuples. The Byzantine PAXOS guarantees that if a correct server removed r tuples then eventually all other correct servers will also remove r tuples. The listener pattern makes each server notify p with the tuples that combine with its template after each removal, so eventually there will be some r for which all correct servers replied to the read operation after r removals. Therefore, the first phase of the algorithm always terminate.

The write-back phase, when necessary, also satisfies wait-freedom since the condition for the client to unblock is the reception of confirmations from a quorum of servers. Since there is always a quorum of correct servers (as $q \leq n-f$) and these servers always reply to a write-back correctly justified, the client cannot block. ■

Lemma 9 *Operation inp is wait-free.*

Proof: The liveness of this operation depends of the modified Byzantine PAXOS. This protocol guarantees that a new leader will be elected until there is some correct leader in a "synchronous" round (i.e., a round in which all messages are exchanged within certain time limits and the leader is not suspected). In this round, the leader will propose a sequence number and a result t for the $inp(\bar{t})$ operation invoked. The properties of Byzantine PAXOS ensure that the sequence number is valid. The result tuple t will be accepted by the servers if it is accepted by $\lceil \frac{n+f}{2} \rceil$ servers. This happens if t the $out(t)$ protocol was correctly executed by the client that inserted t in the space, or if t is correctly justified by a proposal certificate. Therefore, we have to consider three cases:

1. *t was completely inserted.* In this case $q = \lceil \frac{n+2f+1}{2} \rceil$ servers have t and at least $q - f$ (the correct ones) will accept the proposal. Since $q - f \geq \lceil \frac{n+f}{2} \rceil$ always holds for $n \geq 4f + 1$, this proposal will eventually be decided.

2. *t was partially inserted.* If t exists in $\lceil \frac{n+f}{2} \rceil$ servers, it will be accepted and decided as the result of the operation. If t was justified by a proposal certificate showing that t exists in at least $f + 1$ servers, it will be accepted and eventually decided. If neither t exists in $\lceil \frac{n+f}{2} \rceil$ servers nor it is justified by a proposal certificate, this value will not be accepted by sufficiently many servers and a new leader will be elected. This leader will receive the matching tuples of all servers and then, if it is correct, will choose a result that can be justified for the operation.

3. $t = \perp$. This value will be accepted by a server if there is no tuple that matches the given template or if this value is justified by a proposal certificate (if the certificate shows that there is no tuple that exists in $f + 1$ servers). If this value is accepted by $\lceil \frac{n+f}{2} \rceil$ servers, it will be eventually decided.

In all these cases, t will be accepted, and the operation ends. ■

Using all these lemmas we can prove that LBTS is a tuple space implementation that satisfies linearizability and wait-freedom.

Theorem 1 *LBTS is a linearizable wait-free tuple space.*

Proof: Wait-freedom is ensured by Lemmas 7, 8 and 9, so we only have to prove linearizability.

Consider any history H of a system in which processes interact uniquely through LBTS. We consider that H is complete, i.e., all invocations in H have a matching response[17]. In this history a set of tuples T_H are manipulated, i.e., read, inserted or removed. Since there is no interference between operations that manipulate different tuples, we can consider that each tuple t is a different object and for each tuple $t \in T_H$ we denote by $H|t$ the subhistory of H that contains only operations that manipulate t.

For each $H|t$, our tuple uniqueness assumption ensures that there will be only one $\langle out(t)/ack \rangle$ and Lemma 6 ensures that there will be no more than one removal of t in this subhistory. Therefore, $H|t$ contains one *out* operation, zero or more readings of t and at most one *inp* operation with result t.

The proof that LBTS is a linearizable tuple space has three steps. First, we build a sequential history $H'|t$ for each tuple $t \in T_H$ with all sequential operations of $H|t$ preserving their original order. The second step is to order concurrent operations according to the properties of LBTS (stated by Lemmas 1-6). Then we will show that $H'|t$ is according to a sequential specification of a tuple space in which just one tuple is manipulated. Finally, we will use the *locality* property of linearizability (Theorem 1 of [26]): if for all $t \in T_H$, $H|t$ is linearizable, then H is linearizable.

1. For sequential operations, for each tuple $t \in T_H$, Lemmas 2 and 3 show that a $\langle rdp(\bar{t})/t \rangle$ can only occur in $H|t$ after its insertion. Lemmas 3, 4 and 5 show that all $\langle rdp(\bar{t})/t \rangle$ will happen before the removal of t.

2. For each tuple $t \in H$, we will order concurrent operations in $H|t$, obtaining $H'|t$ with all operations in $H|t$. The ordering is done the following way: all $\langle rdp(\bar{t})/t \rangle$ are put after $\langle out(t)/ack \rangle$ (Lemma 2 states that t cannot be read before $\langle out(t)/ack \rangle$, and Lemma 3 states t can be read after $\langle out(t)/ack \rangle$) and before $\langle inp(\bar{t})/t \rangle$ (Lemma 3 states that t can be read before $\langle inp(\bar{t})/t \rangle$ and Lemma 5 states that t cannot be read after $\langle inp(\bar{t})/t \rangle$). The order between different *rdp* operations that return t does not matter since Lemma 4 states that once read, t will always be read until its removal.

3. After ordering the operations in the described way, all histories $H'|t$, for each tuple $t \in T_H$ will begin with an $\langle out(t)/ack \rangle$, will have zero or more $\langle rdp(\bar{t})/t \rangle$ after, and can end with an

[17] If it is not complete, we can extend it with the missing matching responses.

$\langle inp(\bar{t})/t \rangle$. Clearly, this history satisfies all four correctness conditions presented in Section 3, even when operations are executed concurrently in the system. Therefore, for all $t \in T_H$, $H|t$ is linearizable.

4. Using the linearizability locality property, we conclude that H is linearizable.

This means that all histories in which a set of processes communicate uniquely using LBTS are linearizable, so LBTS is linearizable. ■

References

[1] M. Abd-El-Malek, G. Ganger, G. Goodson, M. Reiter, and J. Wylie. Fault-scalable Byzantine fault-tolerant services. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles - SOSP'05*, pages 59–74, Oct. 2005.

[2] I. Abraham, G. Chockler, I. Keidar, and D. Malkhi. Byzantine disk paxos: optimal resilience with Byzantine shared memory. *Distributed Computing*, 18(5):387–408, Apr. 2006.

[3] D. E. Bakken and R. D. Schlichting. Supporting fault-tolerant parallel programing in Linda. *IEEE Transactions on Parallel and Distributed Systems*, 6(3):287–302, Mar. 1995.

[4] R. A. Bazzi and Y. Ding. Non-skipping timestamps for Byzantine data storage systems. In *Proceedings of 18th International Symposium on Distributed Computing - DISC 2004*, pages 405–419, Oct. 2004.

[5] A. N. Bessani, M. Correia, J. da Silva Fraga, and L. C. Lung. Sharing memory between Byzantine processes using policy-enforced tuple spaces. In *Proceedings of 26th IEEE International Conference on Distributed Computing Systems - ICDCS 2006*, July 2006.

[6] A. N. Bessani, J. da Silva Fraga, and L. C. Lung. BTS: A Byzantine fault-tolerant tuple space. In *Proceedings of the 21st ACM Symposium on Applied Computing - SAC 2006*, pages 429–433, 2006.

[7] G. Bracha and S. Toueg. Asynchronous consensus and broadcast protocols. *Journal of ACM*, 32(4):824–840, 1985.

[8] N. Busi, R. Gorrieri, and G. Zavattaro. On the expressiveness of Linda coordination primitives. *Information and Computation*, 156(1-2):90–121, Jan. 2000.

[9] G. Cabri, L. Leonardi, and F. Zambonelli. Mobile agents coordination models for Internet applications. *IEEE Computer*, 33(2):82–89, Feb. 2000.

[10] C. Cachin and S. Tessaro. Optimal resilience for erasure-coded Byzantine distributed storage. In *Proceedings of the International Conference on Dependable Systems and Networks - DSN 2006*, pages 115–124, June 2006.

[11] M. Castro and B. Liskov. Practical Byzantine fault-tolerance and proactive recovery. *ACM Transactions Computer Systems*, 20(4):398–461, Nov. 2002.

[12] M. Castro, R. Rodrigues, and B. Liskov. BASE: Using abstraction to improve fault tolerance. *ACM Transactions Computer Systems*, 21(3):236–269, Aug. 2003.

[13] M. Correia, N. F. Neves, and P. Veríssimo. How to tolerate half less one Byzantine nodes in practical distributed systems. In *Proceedings of the 23rd IEEE Symposium on Reliable Distributed Systems - SRDS 2004*, pages 174–183, Oct. 2004.

[14] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira. HQ-Replication: A hybrid quorum protocol for byzantine fault tolerance. In *Proceedings of 7th Symposium on Operating Systems Design and Implementations - OSDI 2006*, Nov. 2006.

[15] P. Dutta, R. Guerraoui, R. R. Levy, and A. Chakraborty. How fast can a distributed atomic read be? In *Proceedings of the 23rd annual ACM Symposium on Principles of Distributed Computing - PODC 2004*, pages 236–245, July 2004.

[16] C. Dwork, N. A. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–322, 1988.

[17] R. Ekwall and A. Schiper. Replication: Understanding the advantage of atomic broadcast over quorum systems. *Journal of Universal Computer Science*, 11(5):703–711, 2005.

[18] F. Favarim, J. S. Fraga, L. C. Lung, and M. Correia. GridTS: A new approach for fault-tolerant scheduling in grid computing. In *Proceedings of 6th IEEE Symposium on Network Computing and Applications - NCA 2007*, July 2007.

[19] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, Apr. 1985.

[20] J. Fraga and D. Powell. A fault- and intrusion-tolerant file system. In *Proceedings of the 3rd International Conference on Computer Security*, pages 203–218, 1985.

[21] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programing Languages and Systems*, 7(1):80–112, Jan. 1985.

[22] D. Gelernter and N. Carriero. Coordination languages and their significance. *Communications of ACM*, 35(2):96–107, 1992.

[23] D. Gifford. Weighted voting for replicated data. In *Proceedings of the 7th ACM Symposium on Operating Systems Principles - SOSP'79*, pages 150–162, Dec. 1979.

[24] GigaSpaces. GigaSpaces homepage. Avaliable at *http://www.gigaspaces.com/*, 2007.

[25] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programing Languages and Systems*, 13(1):124–149, Jan. 1991.

[26] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programing Languages and Systems*, 12(3):463–492, July 1990.

[27] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programing Languages and Systems*, 4(3):382–401, July 1982.

[28] B. Liskov and R. Rodrigues. Tolerating Byzantine faulty clients in a quorum system. In *Proceedings of 26th IEEE International Conference on Distributed Computing Systems - ICDCS 2006*, 2006.

[29] D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, Oct. 1998.

[30] D. Malkhi and M. Reiter. Secure and scalable replication in Phalanx. In *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems - SRDS'98*, pages 51–60, Oct. 1998.

[31] D. Malkhi and M. Reiter. An architecture for survivable coordination in large distributed systems. *IEEE Transactions on Knowledge and Data Engineering*, 12(2):187–202, Apr. 2000.

[32] J.-P. Martin and L. Alvisi. Fast Byzantine consensus. *IEEE Transactions on Dependable and Secure Computing*, 3(3):202–215, July 2006.

[33] J.-P. Martin, L. Alvisi, and M. Dahlin. Minimal Byzantine storage. In *Proceedings of the 16th International Symposium on Distributed Computing - DISC 2002*, pages 311–325, Oct. 2002.

[34] J.-P. Martin, L. Alvisi, and M. Dahlin. Small Byzantine quorum systems. In *Proceedings of the Dependable Systems and Networks - DSN 2002*, pages 374–388, June 2002.

[35] A. Murphy, G. Picco, and G.-C. Roman. LIME: A coordination model and middleware supporting mobility of hosts and agents. *ACM Transactions on Software Engineering and Methodology*, 15(3):279–328, July 2006.

[36] R. R. Obelheiro, A. N. Bessani, L. C. Lung, and M. Correia. How practical are intrusion-tolerant distributed systems? DI-FCUL TR 06–15, Dep. of Informatics, Univ. of Lisbon, 2006.

[37] R. L. Rivest, A. Shamir, and L. M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[38] F. B. Schneider. Implementing fault-tolerant service using the state machine aproach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, Dec. 1990.

[39] E. J. Segall. Resilient distributed objects: Basic results and applications to shared spaces. In *Proceedings of the 7th Symposium on Parallel and Distributed Processing - SPDP'95*, pages 320–327, Oct. 1995.

[40] Sun Microsystems. JavaSpaces service specification. Avaliable at `http://www.jini.org/standards`, 2003.

[41] T. J. Lehman et al. Hitting the distributed computing sweet spot with TSpaces. *Computer Networks*, 35(4):457–472, 2001.

[42] P. Verissimo, N. F. Neves, and M. P. Correia. Intrusion-tolerant architectures: Concepts and design. In *Architecting Dependable Systems*, volume 2677 of *LNCS*. 2003.

[43] A. Xu and B. Liskov. A design for a fault-tolerant, distributed implementation of Linda. In *Proceedings of the 19th Symposium on Fault-Tolerant Computing - FTCS'89*, pages 199–206, June 1989.

[44] P. Zielinski. Paxos at war. Technical Report UCAM-CL-TR-593, University of Cambridge Computer Laboratory, Cambridge, UK, June 2004.